Code Verification Workflow in CASL

William J. Rider, James R. Kamm, and V. Gregory Weirs
Sandia National Laboratories, Albuquerque NM 87185
September 30, 2010

Overview

Code verification is a well-defined process by which the correctness and accuracy of a software implementation of a numerical algorithm can be evaluated. *Solution verification* is a related but distinct process by which the discretization error is estimated in simulations of interest. In this document a workflow for code verification is presented.

Most numerical methods used to obtain approximate numerical solutions of continuum models have a number of key properties. Among these characteristics is the order-of-accuracy (also called the convergence rate), which is given by the exponent in the power law relating the numerical truncation error to the value of the discret parameter. The most common approach to code verification is to compare the *theoretical* rate of convergence of the numerical method to the *observed* rate produced by an implementation of that method, to gauge the correctness of the implementation.

The procedure by which to provide this measure of correctness is systematic mesh refinement (or variation). The results of this approach are combined with error measurement to produce the observed rate-of-convergence, which is compared with the ideal or theoretical rate-of-convergence of the underlying algorithm. For code verification, the use of an analytical or exact solution to a problem plays a fundamental role in the process by providing an unambiguous fiducial solution.

In summary, the workflow for conducting code verification is the following:

1. Starting with an implementation (i.e., code) that has passed the appropriate level of SQA, choose the executable to be examined.
2. Provide a complete analysis of the numerical method as implemented including accuracy and stability properties.
3. Select the analytical solution(s) for problems to be examined, and provide the analytical solution in a form that allows direct comparison with the numerical solutions and provide the means for computing the errors in the numerical solution.
4. Produce the code input to model the problem(s).
5. Select the sequence of mesh discretizations to be examined for each solution.
6. Run the code and provide the means of producing appropriate metrics to evaluate the difference between the numerical and analytical solutions.
7. Use the comparison to determine the sequence of errors corresponding to the various discretizations.

8. The error sequence allows the determination of the rate-of-convergence for the method, which is compared to the theoretical rate.
9. Using these results, render an assessment of the method's implementation correctness.
10. Examine the degree of coverage of features in an implementation by the verification testing.

In a modern code development environment, this process should be repeatable and available on-demand.

The focus of this document is code verification, which is a necessary prerequisite for solution verification, validation, and uncertainty quantification. These other assessment techniques are only briefly introduced to distinguish them from code verification.

What is Verification?

There are different kinds of Verification:

- *Code verification* – comparing the results of a coded algorithm (i.e., instantiated in software) with an analytical or exact (i.e., "closed form") solution or highly accurate solution obtained by some other means[1], for the purpose of assessing the code.
- *Calculation* or *solution verification* – using the demonstrated convergence properties of the code to estimate numerical errors in solving the model, involving the evaluation of results of the code alone.

It is notable that the credibility of calculation verification is predicated on producing error estimates from a code that have passed appropriate and relevant code verification.

At its core, verification of scientific simulation software both quantifies numerical errors and defines a rigorous basis for believing that quantification. Providing an error estimate for complex problems falls under the purview of *solution* (or *calculation*) *verification*, while providing the rigorous basis for such estimates is achieved with *code verification*. The overall activity of verification is the combination of *both* code and solution verification.

[1] For example, for certain problems governed by specific partial differential equations (PDEs), the equation can be reduced to an ordinary differential equation (ODE), the solution of which can often be obtained very accurately. Such a highly accurate numerical solution of an ODE could be used as the de facto "exact" solution for purposes of verification analysis of a PDE solver for that problem.

- *Software verification* – checking for a correct functioning of the software system on a particular platform.

Such software testing is a critical element of software development. The testing that is closest to code development centers on software engineering techniques, such as unit testing and regression testing, both of which address the correct functioning of software. These types of testing use generic success metrics that apply to almost all classes of software. In contrast, code and solution verification are assessment techniques for software that provides approximate solutions, with metrics specialized to the particular type of algorithm.

There is persistent confusion of verification testing with regression testing: these are completely different testing procedures with completely different goals. Regression testing is a software engineering technique that assesses the robustness of software to frequent changes. Regression tests reduce to a (typically large) collection of relatively simple problems that are executed at a regular (typically frequent) time interval. Regression testing seeks principally to reduce the amount of software rework that is created by the introduction of mistakes in software modifications. This reduction is accomplished by comparing today's code with yesterday's code via execution of the regression test suite. Thus, regression testing targets *software stability*, not *mathematical correctness*.

At this point other popular forms of assessment are described, as is their relationship with code verification and the workflow that is the topic of this document. *Solution verification* estimates numerical errors for a problem of interest based on an assumed relationship between numerical error and resolution (a measure of the discretization parameter); code verification tests whether this assumption is satisfied.

Unlike verification, *validation* tests whether a model is a sufficiently accurate representation of the physical processes in a particular problem. In scientific simulations the *model* refers to the governing equations, which include initial conditions, boundary conditions, constitutive relations, etc. To simulate essentially any nontrivial physical configuration, these equations must be solved computationally, which depends on numerical algorithms, corresponding software implementations, and appropriate use of that software. Therefore, validation entails comparisons of approximate solutions of the governing equations (which are an imperfect representation of the relevant physical processes) to experimental data (which also contain inaccuracies). Quantitative comparison of experiment (having, e.g., physical and diagnostic uncertainties) with simulations (having concomitant modeling, algorithmic, and solution errors) remains a challenging undertaking. It is regular experimental practice to provide error bars showing the degree of uncertainty in physical data, and solution verification can help provide estimates of the numerical error in the simulations.

Every simulation requires a number of inputs to specify the problem to be solved, to

choose among various numerical methods and how they are to be applied, and to provide values for parameters in specific material models. These values may not be known exactly, or may represent some average value that depends on the particular situation. The process of identifying and quantifying the effects of the uncertainty of these simulation inputs on the results of the simulation is called *uncertainty quantification* . The most common way of estimating these effects is to run a (preferably large) number of simulations, in which the values of the inputs are taken from distributions, to determine the corresponding distribution of the results of interest. Unlike code verification, solution verification, or validation, uncertainty quantification deals with the sensitivity of simulation results to how the problem is specified, rather than how well that problem is solved.

Numerical simulations are increasingly used to increase understanding, to "solve" problems, to design devices, vehicles and buildings, and inform high-consequence decision makers. The different assessment techniques address different ways numerical simulations can fail to provide accurate information. Ultimately, only the combined application of all the techniques can provide confidence that the accuracy of the numerical simulation process is adequate for a particular scenario. In practice, code verification is the foundation upon which the other assessment techniques rest. The premise of solution verification is that the code converges at a known rate as the resolution is increased; code verification establishes that this is in fact the case. To compare experimental results to code results, both the experimental and numerical errors must be accurately quantified so they can be accounted for in the comparison. Validation relies on solution verification to provide the numerical error, which in turn relies on code verification. The process of uncertainty quantification accepts the model, in this case the code that produces numerical simulations, as an input. Inferences drawn about the system that the model represents are inherently limited by the accuracy of the model. Code verification ensures the model is correctly implemented and underlies solution verification and validation that quantify the accuracy of the model.

Code Verification

Who does verification? Code developers, mathematicians, and algorithm engineers

Complex simulation software cannot be proven to be mathematically correct. Consequently, the accumulation of quantitative evidence remains the exclusive basis for inferring the mathematical correctness. The practical view is that this evidence is accumulated over time. This accumulation occurs throughout the on-going processes of code development as well as during the subsequent code usage. Thus, the results of verification analyses are affected by the manner in which software is generated and the proper specification/execution of the verification problem. While the former is the purview of code developers and algorithm designers, responsibility for the latter falls upon those who describe the verification problem both in documentation and in actual code input. For those using a code to conduct

analysis, their responsibility is to act mindfully regarding the quality of the code, and the relevance of the testing to their problems of interest. At best, they should act as advocates for quality control measures such as rigorous, extensive verification because it supports any calculations where the code is used.

Verification evidence emerging from code development is generated by software engineering processes applied during that development, and by the specific testing practices employed by the development team. Code usage evidence is a more nuanced and diverse body of information that emerges from a heterogeneous group of users. Testing executed under the umbrella of code development is not restricted to the verification approaches discussed in this document. Unlike other testing procedures applied by code developers (including, e.g., unit tests and the restricted cases applied in regression testing), verification test problems also are relevant to code users.

Verification test suites can be implemented, managed, and applied by code developers just like regression test suites. The major differences are: (i) the development and execution of verification test suites takes more resources (people, computers, time); (ii) the time interval of execution of a verification test suite will be different than for regression testing; and (iii) the direct methods for comparing today's regression test suite results versus yesterday's baseline should be replaced by greater human involvement in judging the quality of the verification tests where possible. On the other hand, we believe that subjective judgment is ultimately always associated with the quality of software at some fundamental level. For example, a verification study rarely produces results that exactly produce the theoretical convergence rate; in fact, the observed rates can vary greatly. The judgment of whether the result is close enough to expectations remains a largely expert matter. Improvement of this state of affairs is a present topic of research. This increased human element required to assess the execution of verification tests emphasizes that an important value of verification tests is their use in engaging the user community around a code.

What is done in verification? Compare code solutions with analytical solutions.

To conduct a verification analysis, one must have (i) a clear statement of the problem with sufficient information to run a computer simulation, (ii) an explanation of how the code result and benchmark solution are to be evaluated, and (iii) a description of the acceptance criteria for a specific simulation code's results on a particular problem; in the absence of such criteria, a process for acceptance is necessary. These concepts are adapted from the notion of a "strong sense verification benchmark," proffered by Oberkampf and Trucano [Obe07], and are intended to reduce the ambiguity of verification problem statements and bolster the value of verification analyses.

Here, the problem statement should include not only a mathematical description of the problem but also a discussion of the processes modeled (i.e., what the problem tests), the initial and boundary conditions, additional numerical information (e.g., convergence criteria used), the principal code features tested, and the nature of the test. This latter element is addressed in [Obe07], with a set of different categories of benchmarks. Kamm et al. [Kam08] provide examples of such problem statements.

Given the complexity of many problems of interest, however, such problem descriptions may still not be definitive, i.e., there may remain unspecified choices in problem set-up that the code analyst must make. Nevertheless, such a description provides a starting points for setting up the problem as well as a touchstone against which one can compare descriptions of the "identical" verification problem, run by different analysts with different simulation codes on different platforms at different times.[2] In the written description and analysis of verification problems, it is imperative that researchers describe *as thoroughly as possible* the complete specification and set-up of the problem (up to including the code input deck in the written report).

Why is verification done? To make sure that a model in a code is implemented correctly.

The outcome of code verification analyses provides hard evidence of mathematical consistency—or inconsistency—between the mathematical statements of the physics models and their discrete analogues as implemented with numerical algorithms in the simulation codes. The necessity of code verification must be emphasized. In the absence of confirmatory verification evidence, "good agreement" of calculations with experimental data could be accidental, i.e., "the right answer for the wrong reasons."

A common confusion with regard to code verification is associated with software quality assurance, which is a vital, but primarily unrelated activity and discipline in its own right. Code verification typically flourishes in a development culture focused on high quality software development, but good code verification practices are neither necessary nor sufficient for good SQA practices—and vice-versa. Each area of expertise should be independently developed and supported, although the practice of each is mutually self-supporting.

The purpose of scientific simulation software differs from that of much commercial software, which is often intended to provide exact solutions to problems that actually have exact solutions (e.g., spreadsheets) or to generate results for problems

[2] Analyses containing the possible differences mentioned here differ markedly from "code comparison" exercises, problematic aspects of which are pointed out by Trucano et al. [Tru03]. Verification acceptance criteria should be sufficiently forgiving to allow small variation associated with imperfect specification of verification problems.

that have subjectively defined goals (e.g., image processing, word processors). Verification is needed for scientific simulation codes because that software is designed to produce approximate solutions to mathematical problems for which (i) the exact solution is not known and (ii) knowledge of the error is potentially as valuable as knowledge of the solution, per se. Due to these distinguishing and critical aspects of scientific simulation codes, software quality practices from the broader industry (e.g., regression testing) are necessary but not sufficient for high-consequence scientific simulation codes.

Verification analysis of scientific simulation codes is an example of the assessment of a complex system for which the systematic gathering of appropriate evidence is required. While tests may demonstrate that software is manifestly incorrect, there is no clear-cut procedure with which to "prove" unambiguously that software behavior is, indeed, correct. Thus, the process by which relevant verification evidence is generated and interpreted requires knowledge of the entire simulation and analysis chain. Such knowledge includes understanding of:

- the system being simulated (e.g., the relevant physics, physics models, and these models' representations in mathematical equations);
- the nature of the simulation (including the algorithms used to obtain approximate solutions to the mathematical equations, these algorithms' limitations, the associated numerical analysis, and the software implementation of those algorithms); and
- the process by which the code results are analyzed in the verification process (including, e.g., theory, implementation, and interpretation of convergence analysis).

This body of knowledge is both large and multi-faceted; consequently, the determination of appropriate of verification problems requires guidance from and consensus among experts in each of these fields.

Decision makers and code analysts should bear in mind that simulation software represents intricate numerical algorithms coupled with a complicated hardware/system-software platform. Said another way, code users and their customers should recognize that simulation software is *not* a "physics engine" that generates instantiations of physical reality. Hence, documented, quantitative verification analysis is a necessary component for developing code confidence and credibility.

What is hard? The analytical structure of solutions is not always known. The *verification* studies are quite tedious.

The difficulties of verification are many and subtle. The first issue for the practitioner to confront is typically the lack of general, complex and application-relevant analytical solutions. Generally speaking, analytical solutions are only available for simple problems with simple geometries, for simple physics, often with idealized (or singular) boundary/initial conditions. Even with relative simplicity,

the analytical structure of exact solutions can be quite complex. In many cases the evaluation of analytical solutions—in and of itself—is difficult and requires acute attention to software quality, numerical algorithms, convergence, and other details.

Secondly, the conduct of verification is, frankly, fraught with tedium. The code being verified must be run repeatedly is a controlled manner. Many details such as mesh, time step, time at which results are stored, etc., must all be scrupulously attended to. The results must be stored and compared in a compatible manner. The analytical solution must be available in a manner consistent with the numerical solution, and the comparison must be computable. The myriad bookkeeping details that are manifest in the activity require substantial discipline.

The third issue is the interpretation of results from verification studies. The correctness of the results depends upon the nature of the mathematical theory associated with both the governing equations and their numerical solution. This theory often differs on the basis of solution character (i.e., smoothness), time, nonlinearity, approximation detail, computer implementation, precision, etc., all of which must be known and accounted for. It is often the case that mathematical results for numerical methods are asymptotic in nature, and the actual numerical solutions are not computed in the same asymptotic range. This difference causes some degree of uncertainty in the results, which may or may not be important.

Verification Techniques

The workhorse technique for verification is systematic mesh refinement (or coarsening). A fundamental expectation for a numerical method is the systematic reduction in solution error as the characteristic length scale associated with the mesh is reduced. In the asymptotic limit where the mesh length scale approaches zero, the method should produce a rate of convergence equal to that defined by numerical analysis (often obtained with the aid of the Taylor series expansion). To conduct analysis using this technique, a sequence of grids with different intrinsic mesh scales are used to compute solutions and their associated errors. The combination of errors and mesh scales can then be used to estimate the rate of convergence for the method in the code. In order to estimate the convergence rate a minimum of two grids are necessary (giving two error estimates, one for each grid).

Another tool used in verification are error estimators. These methods are commonly derived in the finite element method (FEM) in solving elliptic partial differential equations. The best-known method is the Zienkiewicz-Zhu error estimator [Zie92]. One can use the error estimate constructively to drive adaptive mesh refinement, or to produce an estimate of the error on a given mesh. While the error estimation is produced, this approach does not necessarily produce the sort of evidence basis for code correctness because the rate of convergence defined theoretically is not verified in the process.

Another technique is the method of manufactured solutions (MMS)[Sal00,Roa02]. MMS bypasses the necessity of solving a problem with an analytical solution. With MMS, a closed-form of a solution is posited initially and, using the governing equations, a source term is derived that can be used by the code to "drive" the computed solution to the defined solution. In this case, the other verification techniques can be utilized to evaluate errors, which can then be used to check for error magnitude and scale dependent behavior (i.e., convergence rate).

Another method that FEM practitioners use for testing code correctness is the so-called patch test [Zie97]. In a real sense, the patch test is a necessary but not sufficient condition for code correctness. The patch test usually involves a constant or linear solution to a simple, fundamental problem (such as the stress field in a rectangular domain with classic boundary conditions). The linearity of the field should be *exactly* reproduced by linear finite elements. A constant field fulfills the same criteria, but at a lower order. The patch test does not prove the order of convergence for a method, but rather shows a completeness of approximation for a reduced set of solutions.

Choice of Metrics

Several metrics can be used to conduct code or calculation verification. One seeks to evaluate the quantitative difference between two sets of numbers, where each set corresponds to some aspect of the solution. Generally speaking, one should employ the metric that follows naturally from the function space in which numerical analysis proofs of convergence are conducted.

Practically speaking, for simple scalars (say, the value of some scalar property, e.g., temperature, at a particular location at a specified time), the absolute value of the difference between two values is the obvious choice. This notion generalizes naturally to higher dimensional cases involving the difference between (discrete) function values from the computational mesh. As examples, these values could be, e.g., a time series of temperature at a particular location (the relevant computational mesh being in time) or, say, the pressure field over a specified, fixed three-dimensional volume at a specified time (the relevant computational mesh being in space). In such cases, one often uses the familiar "p-norm" of functional and numerical analysis. The p-norm of the function g is given by

$$\| g \|_p \equiv \left(\int_a^b |g(x)|^p \, dx \right)^{1/p}. \tag{1}$$

For example, for finite volume methods applied to discontinuous functions, the use of the 1-norm is recommended, while, say, properties of inherently smooth functions are most appropriately measured in the energy or 2-norm. It is can be enlightening to evaluate several norms, e.g., 1-, 2-, and ∞-norms, where, following from the equation above,

$$\| g \|_\infty \equiv \max_{x \in [a,b]} \left| g(x) \right| \ . \tag{2}$$

In the following, we use the double-bar notation "||" without a subscript to denote any appropriate norm.

Asymptotic Convergence Analysis

The axiomatic premise of asymptotic convergence analysis is that the computed difference between the reference and computed solutions can be expanded in a series based on some measure of the discretization of the underlying equations. Taking the spatial mesh as the obvious example, the ansatz for the error in a 1-D simulation is taken to be

$$\left\| g^{\text{ref}} - g^{\text{comp}} \right\| = A_0 + A_1 (\Delta x)^\alpha + o\left((\Delta x)^\alpha \right) \ . \tag{3}$$

In this relation, g^{ref} is the reference solution, g^{comp} is the computed solution, Δx is some measure of the mesh-cell size, A_0 is the zero-th order error, A_1 is the first order error, and the notation "$o\left((\Delta x)^\alpha \right)$" denotes terms that approach zero faster than $(\Delta x)^\alpha$ as $\Delta x \to 0^+$. For consistent numerical solutions, A_0 should be identically zero; we take this to be the case in the following discussion. For a consistent solution, the exponent α of Δx is the convergence rate: $\alpha = 1$ implies first-order convergence, $\alpha = 2$ implies second order convergence, etc.

Assume that the calculation has been run on a "coarse" mesh (subscript c), characterized by Δx_c, which we hereafter also denote as Δx. The error ansatz implies:

$$\left\| g^{\text{ref}} - g_c^{\text{comp}} \right\| = A_1 (\Delta x)^\alpha + \cdots . \tag{4}$$

We further assume that we have computational results on a "fine" mesh Δx_f (subscript f), where $0 < \Delta x_f < \Delta x_c$ with $\Delta x_c / \Delta x_f \equiv \sigma > 1$. In this case, the error ansatz implies:

$$\left\| g^{\text{ref}} - g_f^{\text{comp}} \right\| = \sigma^{-\alpha} A_1 (\Delta x)^\alpha + \cdots . \tag{5}$$

Manipulation of these two equations leads to the following explicit expressions for the quantities α and A_1:

$$\alpha = \left[\log\left\|g^{\text{ref}} - g_c^{\text{comp}}\right\| - \log\left\|g^{\text{ref}} - g_f^{\text{comp}}\right\|\right] \Big/ \log\sigma \ , \tag{6}$$

$$A_1 = \left\| g^{\text{ref}} - g_c^{\text{comp}}\right\| \Big/ (\Delta x)^{\alpha} \ . \tag{7}$$

These two equalities are the workhorse relations that provide a direct approach to convergence analysis as a means to evaluating the order of accuracy for code verification.

In the case of calculation verification, one does not have an exact solution and, instead, turns to a finely zoned calculation to serve in place of the exact solution. In this case, the convergence rate can be expressed as

$$\alpha = \left[\log\left\|g_f^{\text{comp}} - g_c^{\text{comp}}\right\| - \log\left\|g_f^{\text{comp}} - g_m^{\text{comp}}\right\|\right] \Big/ \log\sigma \ , \tag{8}$$

where the subscript *m* here denotes values on a "medium" mesh, i.e., one for which $0 < \Delta x_f < \Delta x_m < \Delta x_c$ with $\Delta x_c / \Delta x_m \equiv \sigma > 1$.

Verification Subtleties

There are several subtle but important—and, in some cases, open—issues associated with the estimation of the quantities mentioned above. While the following topics may be considered by some to be arcane, they should be borne in mind by those devising and conducting verification analyses, as well as by code analysts.

• Nondimensionalization The above discussion of the error ansatz and the associated convergence parameters contain no assumptions regarding the dimensions of the associated variables. Consequently, parameters in the resulting scaling relations (e.g., Eq. 4) may have inconsistent units. One way to avoid this issue is to nondimensionalize all quantities prior to conducting such an analysis. For example, one can choose representative quantities *G* and *X* with which to nondimensionalize the computed quantity *g* and the representative mesh scale Δx:

$$g' \equiv g/G \quad \text{and} \quad \Delta x' \equiv \Delta x/X \ . \tag{9}$$

The nondimensional error ansatz is posited to be

$$\left\| g'^{\text{ref}} - g_c'^{\text{comp}}\right\| = A_1(\Delta x')^{\alpha} + \cdots, \tag{10}$$

where all terms in this equation are now dimensionless. In this case, care must be taken to nondimensionalize consistently throughout the analysis, and to properly

dimensionalize results, e.g., if one were to use this relation to estimate errors at another mesh size.

• Dimension For problems in multiple space dimensions (e.g., 2-D Cartesian (x,y)), the spatial convergence analysis described above can be assumed to carry over directly, such that, e.g., the ansatz of Eq. 3 follows identically. That is, one typically does not assume separate convergence rates in separate coordinates. This is a reasonable assumption in almost all cases; the exception is time-convergence, since the time-integration scheme for a PDE may be of different order than the spatial integrator. For a more thorough discussion and examples of combined space-time convergence, see [Hem05, Tim06a]

• Frame Spatial convergence analysis is idealized to refer to a fixed mesh, i.e., the Eulerian frame. Approaches have been taken to extend convergence analysis simplistically to the Lagrangian frame (e.g., [Kam03]). More sophisticated approaches, however, are needed; for example, since the fundamental Lagrangian equations are discretized with respect to *mass* and not *space*, an error ansatz analogous to Eq. 3 with Δx replaced by Δm would be more faithful to the underlying formulation.

• Non-uniform Meshes The intention behind the expression "Δx" in Eq. 3 is that it is a meaningful measure of the characteristic length-scale of mesh cells of the discretized equations. If either adaptive mesh refinement (AMR) or an arbitrary Lagrangian-Eulerian (ALE) approach is used, however, such a quantity—if one exists—is likely to change during the course of a calculation. Again, straightforward approaches for non-uniform and AMR meshes have been examined (e.g., [Li05b]), but these are topics of open research.

• Norm Evaluation The expression for the norm in Eq. 1 is appropriate, e.g., for Cartesian geometries. This term must be appropriately modified for non-Cartesian geometries. For example, for 1-D spherically symmetric calculations, the integral of the norm is properly expressed as

$$\| g \|_p \equiv \left(\int_a^b |g(r)|^p \, dV(r) \right)^{1/p} = \left(\int_a^b |g(r)|^p \, 4\pi r^2 dr \right)^{1/p}. \quad (11)$$

In general, when evaluating the norm one must be mindful of the domain of the integral as well as any symmetries associated with the problem.

• Norm Evaluation and Exact Solutions The definition for the norm in Eq. 1 suggests a simple evaluation of this expression. As previously suggested, in 1-D one might evaluate the norm as:

$$\| g^{\text{ref}} - g^{\text{comp}} \|_1 \equiv \int_a^b \left| g^{\text{ex}}(x) - g^{\text{comp}}(x) \right| dx \approx \sum_{i=1}^{N} \left| g^{\text{ex}}(x_i) - g^{\text{comp}}(x_i) \right| \Delta x_i \quad (12)$$

Such an expression, while notionally correct, can obscure important aspects of the computational algorithm. Finite volume discretizations, which are used in many Eulerian and Lagrangian algorithms, provide computed values $g^{\text{comp}}(x_i)$ that are *not* point values but are actually *averages* over the computational cell. Despite the associated inaccuracy, one often uses point-values of the reference solution and cell-averaged computed values in numerical evaluation of expressions such as Eq. 12. Verification lore for the Riemann problem of 1-D hydrodynamics and numerical results with high-resolution numerical schemes for many calculations suggest that the discrepancy incurred by this assumption is small (say, that it does not affect the leading significant figure of the calculated convergence rate). Rigorous numerical evidence with such a numerical scheme for the Cog-8 problem is given by Timmes et al. [Tim06b], who show that the leading digit of the convergence rate is the same for both point values and cell-averaged values, consistent with anecdotal notions. It is reasonable to anticipate that such results (i.e., that this discrepancy is small) may depend on the particular numerical scheme used.

• Norm Evaluation and Interpolation on Different Meshes The expression for the convergence rate α in the calculation verification (Eq. 8) implies a direct comparison of computed solutions on two different meshes. The analogous expression (Eq. 6) for code verification requires an indirect comparison of computed solutions on different meshes. To evaluate the differences of two calculations, a common mesh is required; this begs the question: should one extrapolate ("restrict") fine-mesh values to the coarse mesh, or interpolate ("prolong") coarse-mesh values onto the fine mesh? Margolin and Shashkov [Mar08] provide a rationale for the former: "…by moving each of the simulation results to the coarsest mesh, we average out the smaller scales and eliminate them as a source of error in studying convergence, thus isolating the discretization error." The detailed manner by which one should move solutions between different meshes remains an open research area. Particular attention should be paid to accurately interpolating solutions near discontinuities.

Example — Analysis of a Linear Elastic Test Problem

In this section, we discuss an example of code verification analysis. The code in question is the Los Alamos National Laboratory RAGE code [Git08]. This code has been developed under the NNSA ASC program, and in accordance with modern SQA procedures. The problem examined is the well-known Blake problem of solid dynamics, which describes the impulsively generated propagation of a spherically symmetric stress wave in a linearly elastic solid [Ald02, Bla52, Hut05, Sha42]. The written report for this analysis [Kam09] contains the analytical form of the exact solution together with RAGE input deck for this problem. As discussed in that report, a set of five different discretizations were considered, with a total of 100,

200, 400, 800, 1600 uniformly-spaced radial zones on a mesh over the domain 0 < r < 100 cm. The computed pressure field is shown, together with the analytic solution, in Fig. 1. Specialized software was used to evaluate the appropriate metrics to compare numerical and analytical solutions, yielding a set of quantitative error estimates for the pressure on each mesh. From these values, the rate of convergence for the pressure was determined and shown to lie between 0.6 and 0.8; these values were compared with the known convergence rate of unity for the RAGE pure hydrodynamics algorithm. This was deemed to have passed the verification criterion for this problem. Additionally, these results led to the speculation that this discrepancy may due to way in which the impulsive boundary condition driving this problem was implemented, either in the code or the input deck. This discrepancy highlights one of the difficulties of nontrivial code verification problems, viz., these problems often capture singular behavior and, therefore, require specialized initial or boundary conditions. This code verification analysis covers the fundamental 1D spherically symmetric hydrodynamics algorithm in the code, as well as the linear elastic response model and the pressure boundary condition.

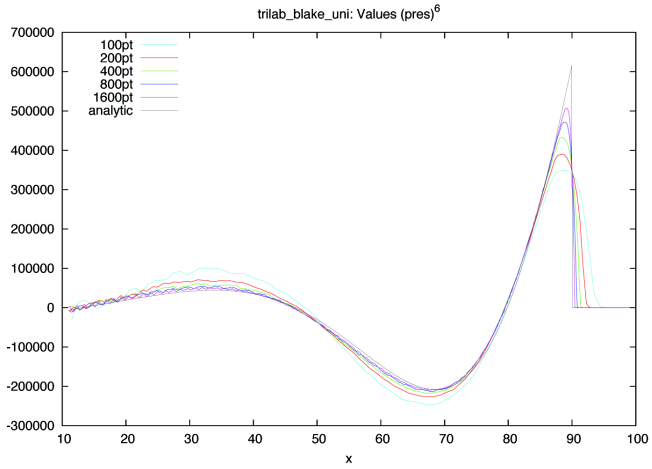

Figure 1. Computed and analytic values of the pressure field at t=1.0 s for the Blake problem. The numbers in the upper left-hand corner correspond to the number of uniform radial zones on 0 < r < 100 cm.

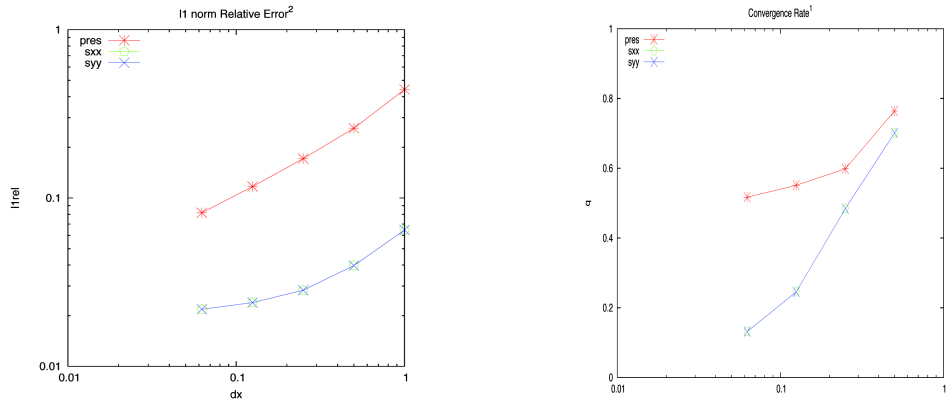

Figure 2. The red symbols and line represent the relative error between exact and analytic solution (left) and inferred convergence rate (right) for the pressure calculations in Fig. 1.

Detailed Workflow

Here, we expand on the details of the workflow. The steps described below are by no means exhaustive, but rather define a standard workflow to be conducted by the code team (developers and testers). Ideally, the code verification process should be conducted regularly (as well as on demand) so that incorrect implementations impacting mathematical correctness are detected as soon as possible. The general consensus in software development is that the cost of bugs is minimized if they are detected as close as possible to their introduction. To start, the verification workflow exists within a broader verification-validation-uncertainty quantification process, which is captured to some extent in Figure 3. This figure depicts the "standard" view of V&V as an activity. It is notable that the code verification is merely a single line on the left hand side of the diagram.

This procedure assumes that the code team is using a well-defined software quality assurance (SQA) process, and the code verification is integrated with this activity. Such SQA includes source code control, regression testing, and documentation, together with other project management activities. For consistency and transparency, we recommend performing the code verification in the same manner and using the same type of tools as other SQA processes.

1. **Starting with an implementation (i.e., code) that has passed the appropriate level of SQA, choose the executable to be examined.** Code verification is a resource-intensive activity involving substantial effort to perform. Code verification should be applied to the same version of the code that analysts would use for any important application. The notion that verification and validation should be applied to the same code is important to keep in mind. This process should be applied to the specific version of the code used throughout the entire V&V UQ activity.

2. **Provide a complete analysis of the numerical method as implemented including accuracy and stability properties.** The analysis should be conducted using any one of a variety of standard approaches. Most commonly, the Von Neumann-Fourier method could be employed. For nonlinear systems, the method of modified equation analysis can be used to define the expected rate and form of convergence. The form and nature of the solution being sought can also influence the expected behavior of the numerical solution. For example, if the solution is discontinuous, the numerical solution will not achieve the same order of accuracy as for a smooth solution. Finite element methods can be analyzed via other methods to define the form and nature of the convergence (including the appropriate norm for comparison).

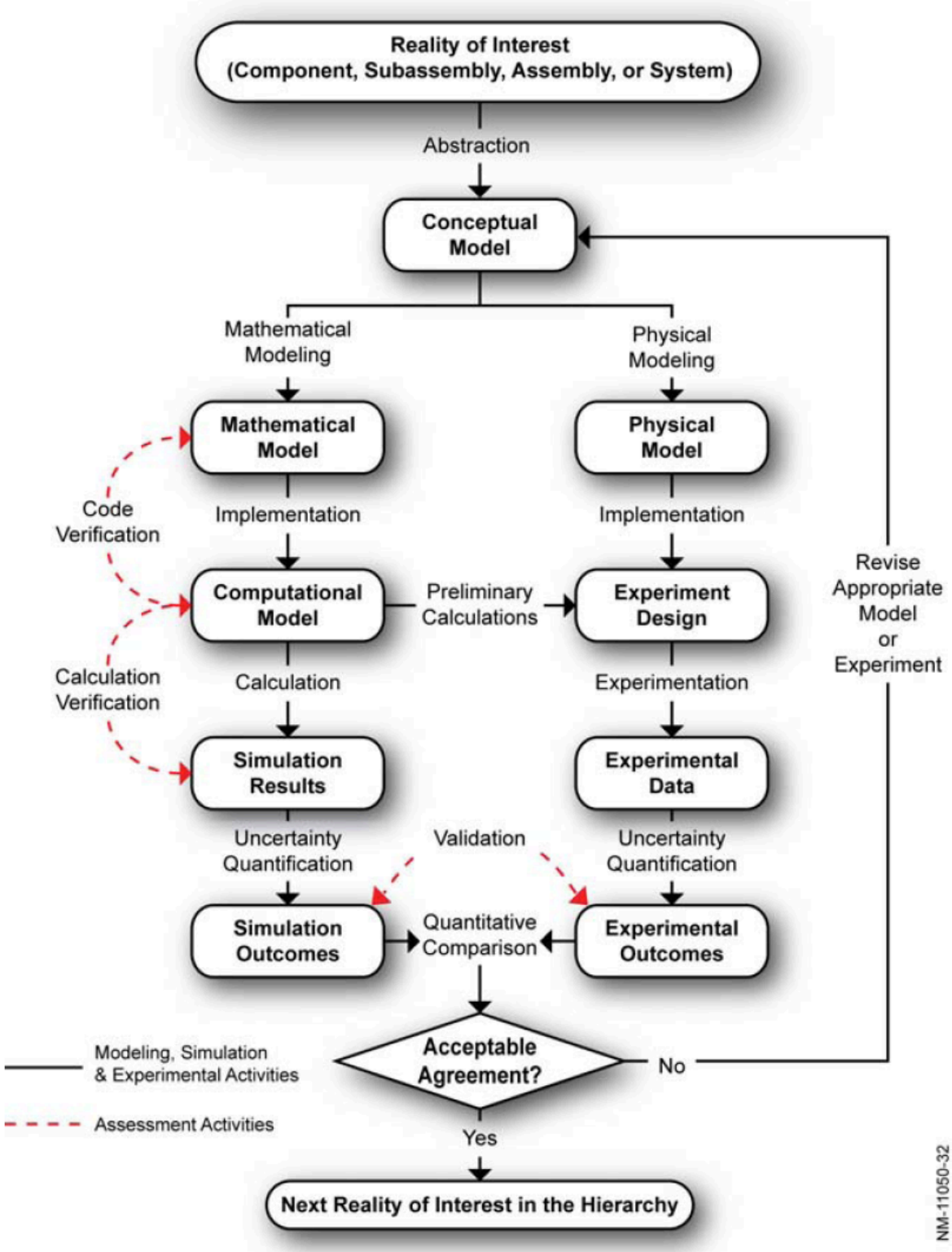

Figure 3. The overall Verification & Validation Flowchart taken from the ASME guide on V&V for Solid Mechanics [Sch06]. This document only covers the loop on the upper left labeled "code verification".

3. **Select the analytical solution(s) for problems to be examined, and provide the analytical solution in a form allowing direct comparison.** Code verification depends upon exact solutions to compute errors in numerical solutions (or "closed form" solutions that can be evaluated to high precision). The determination of analytical solutions is a difficult, the availability of such exact solutions is limited, and the coding of the corresponding exact solution software can be time-consuming. As an alternative, the method of manufactured solutions (MMS) [Knu03,Roy05] can be used, in principle, to produce general solutions to "arbitrarily" complex physics. MMS carries a concomitant complexity in software implementation that must be managed within the confines of the SQA procedures. Finally, the means of comparison between the analytical and numerical solutions must be provided (generally in advance).

4. **Produce the code input to model the problem(s) for which the code verification will be performed.** Each problem is run using the code's standard modeling interface as for any physical problem that would be modeled. It can be a challenging task to generate code input that correctly specifies a code verification problem; e.g., special routines to generate particular initial or boundary conditions that drive the problem may be required, and these routines must be correctly interfaced to the code. It is advisable to consider the complexities and overhead associated with such considerations prior to undertaking such code verification analyses.

5. **Select the sequence of discretizations to be examined so each solution.** Verification necessarily involves convergence testing, which requires that the problem be solved on multiple discrete representations (i.e., grids or meshings). This is consistent with notions associated with *h*-refinement, although other sorts of discretization modification can be envisioned. The mathematical aspects of verification are typically most conveniently carried out if the discretizations are factors of two apart.

6. **Run the code and provide of means of producing appropriate metrics to compare the numerical and analytical solution.** The solutions to the problem are computed on the discretizations. The solutions—both numerical and exact—are compared through well-defined metrics. Most commonly and as discussed above, these metrics take the form of norms (i.e., *p*-norms such as the $L2$ or energy norm). The selection of metrics is inherently tied to the mathematics of the problem and its numerical solution. The metrics can be computed over the entire domain, subsets of the domain, surfaces or specific points. The domain over which the metrics are evaluated and the analysis is conducted must be free of any spurious solution features (due, e.g., waves erroneously reflected from computational boundaries).

7. **Use the comparison to determine the sequence of errors in the discretizations**. Using the well-defined metrics for each solution, the error

can be computed for each discrete representation. Ideally, there will be a set of metrics available, providing a more complete characterization of the problem and its solution.

8. **The error sequence allows the determination of the rate-of-convergence for the method, which is compared to the theoretical rate.** With a sequence of errors in hand, the demonstrated convergence rate of the code for the problem is estimated. The theoretical convergence rate of a numerical method is a key property. Verification relies upon comparing this rate to the demonstrated rate of convergence. Evidence supporting verification is provided when the demonstrated convergence rate is consistent with the theoretical rate of convergence. This can be a difficult inference to draw, because the theoretical rate of convergence is a limit reached in an asymptotic sense, which cannot be reached in for any finite discretization. As a consequence, there are unavoidable deviations from the theoretical rate of convergence, to which judgment must be applied.

9. **Using the results, render an assessment of the method's implementation correctness.** Based on the discrete solutions, errors, and convergence rate, a decision on the correctness of a model can be rendered. This judgment is applied to a code across the full suite of verification test problems.

    a. The assessment can be positive, that is, the convergence rate is consistent with the method's expected accuracy.
    b. The assessment can be negative, that is, the convergence rate is inconsistent with the method's expected accuracy.
    c. The assessment can be inconclusive, that is, one cannot defensibly demonstrate clearly uniform consistency or inconsistency with the method's expected accuracy. For example, the convergence rate is nearly the correct rate, but the differences between the expected rate and the observed rate is uncomfortably large, potentially indicating a problem.

10. **Examine the degree of coverage of features in an implementation by the verification testing.** Code verification is inherently limited in scope by being based on the availability of analytical solutions. MMS can often help to mitigate this issue to some extent. The intent of code verification is to cover the code's capabilities as broadly as possible. Consequently, the coverage of code features should be documented and tracked [Ste05].

Figures 4a,b show the entire process in diagrams that conceptually expand the line for code verification in Figure 3. We repeat our belief that this process should be repeatable and available on-demand. As we noted in the introduction to this section, having the code verification is integrated with the ongoing SQA activity and tools can greatly facilitate this essential property.

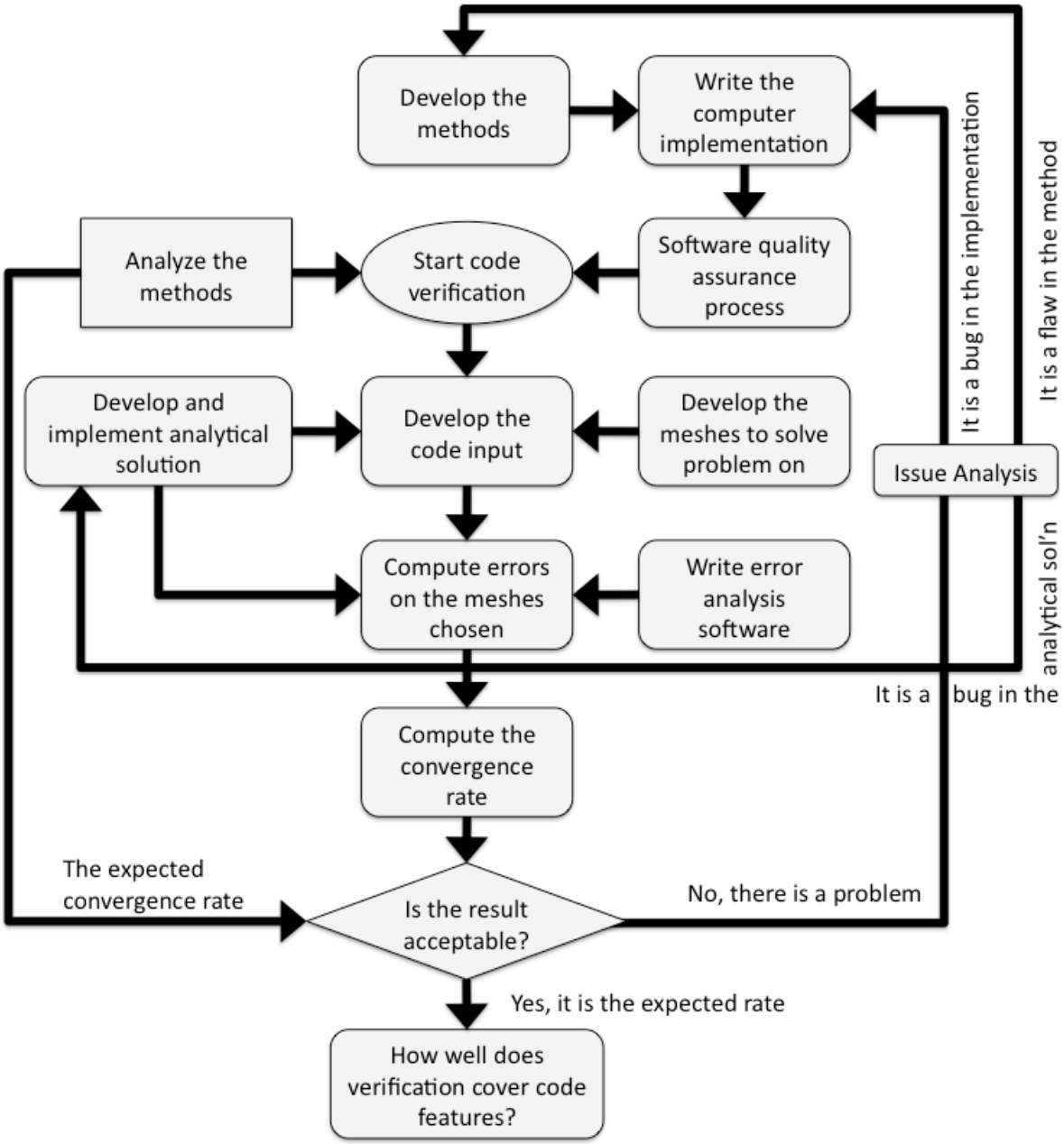

Figure 4(a). The flowchart version of the list of activities is shown for code verification, which can be interpreted as an expansion of the simple expression of this activity.

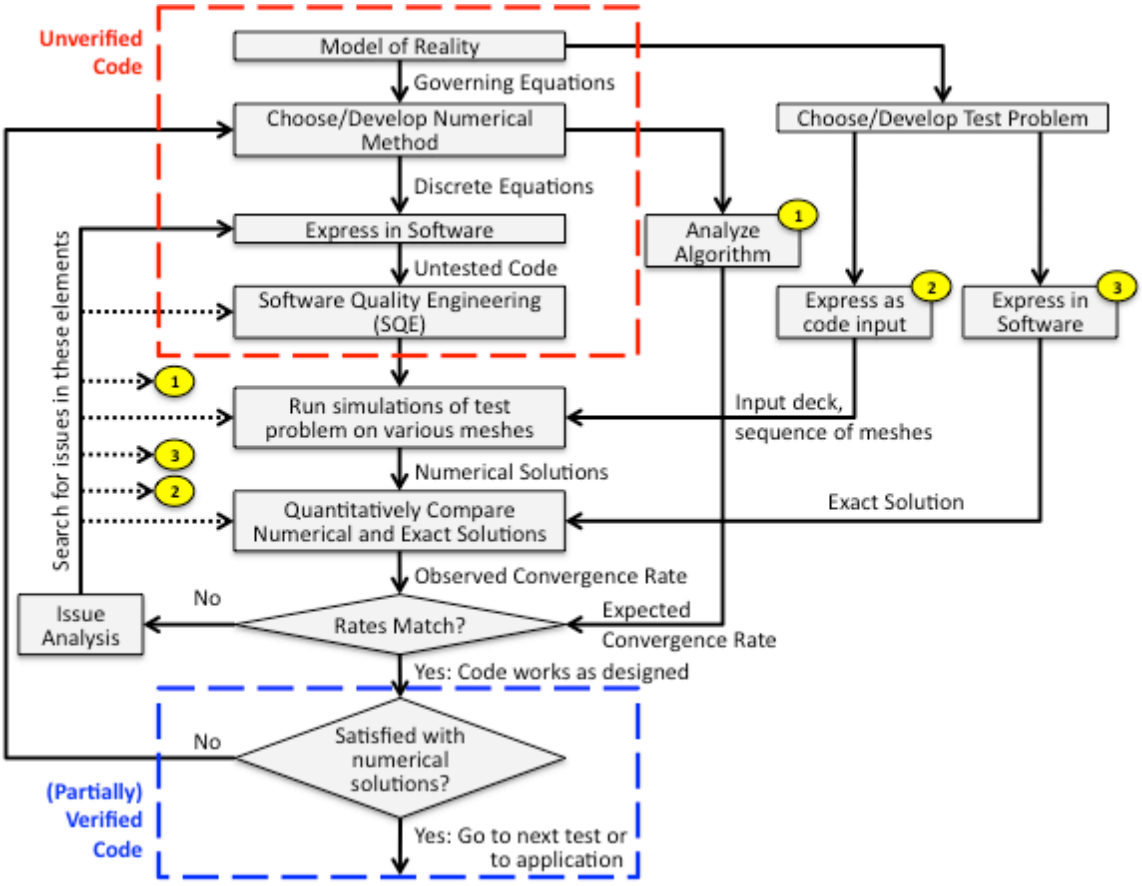

Figure 4(b). The flowchart version of the list of activities is shown for code verification, which can be interpreted as an expansion of the simple expression of this activity.

Conclusions and Recommendations

In this document, we have described the concept of verification, a well-defined process by which the correctness of the implementation of a numerical algorithm in scientific software can be evaluated. We have focused on code verification, an analysis method that quantitatively compares the theoretical order of accuracy of a method with the empirical order of accuracy, which is estimated from error measurements based on code output and analytical ("exact") solutions. We have provided a detailed workflow for conducting code verification.

While this approach to verification is well codified and widely used, there remain details of these analyses that can be difficult to resolve. Most verification cases encountered by the code developer will be standard; however, difficult cases almost always arise eventually. Unless the analyst has chosen exceedingly simple problems, each particular verification problem will likely present its own challenges that will require insight, innovation, and determination on the part of the analyst to resolve. Despite these obstacles, verification is a necessary part of the "due diligence" of a scientific code development project and an essential element in

producing high quality code that can be used for high consequence analysis and decision-making.

Acknowledgement

Sandia National Laboratories is a multi-program laboratory operated by Sandia Corporation, a wholly owned subsidiary of the Lockheed Martin Company, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[Ald02] Aldridge, D. F., *Elastic Wave Radiation from a Pressurized Spherical Cavity*, Sandia National Laboratories Report SAND2002-1882 (2002).

[Bla52] Blake, F. G., "Spherical Wave Propagation in Solid Media," *J. Acoust. Soc. Am.*, **24**, pp. 211–215 (1952).

[Bro06] Brock, J. S., Kamm, J. R., Rider, W. J., Brandon, S. T., Woodward, C., Knupp, P., and Trucano, T. G., *Verification Test Suite for Physics Simulation Codes*, Los Alamos National Laboratory report LA-UR-06-8421.

[Fic74] Fickett, W., and Rivard, W. C., *Test Problems for Hydrocodes*, Los Alamos Scientific Laboratory report LA-5479 (1974).

[Git08] Gittings, M., Weaver, R., Clover, M., Betlach, T., Byrne, N., Coker, R., Dendy, E., Hueckstaedt, R., New, K., Oakes, W. R., Ranta, D., and Stefan, R., "The RAGE radiation-hydrodynamics code," *Comput. Sci. Disc.* , **1**, p. 015005 ff., (2008).

[Hem05] Hemez, F. M., *Non-Linear Error Ansatz Models for Solution Verification in Computational Physics*, Los Alamos National Laboratory report LA-UR-05-8228 (2005).

[Hut05] Hutchens, G. J., *An Analysis of the Blake Problem*, Los Alamos National Laboratory Report LA-UR-05-8738 (2005).

[Kam03] Kamm, J., Brock, J., Rousculp, C., and Rider, W., *Verification of an ASCI Shavano Project Hydrodynamics Algorithm*, Los Alamos National Laboratory report LA-UR-03-6999 (2003).

[Kam08] Kamm, J. R., Brock, J. S., Brandon, S. T., Cotrell, D. L., Johnson, B., Knupp, P., Rider, W. J., Trucano, T. G., and Weirs, V. G., "Enhanced Verification Test Suite for Physics Simulation Codes," Los Alamos National Laboratory report LA-14379, Lawrence Livermore National Laboratory report LLNL-TR-411291, Sandia National Laboratories report SAND2008-7813 (2008).

[Kam09] Kamm, J. R, Ankeny, L. A., "Analysis of the Blake Problem with RAGE," Los Alamos National Laboratory report, LA-UR-09-01255 (2009).

[Knu03] Knupp, P., and Salari, K., *Verification of Computer Codes in Computational Science and Engineering*, Chapman & Hall/CRC, Boca Raton, FL (2003).

[Knu07] Knupp, P., Ober, C., and Bond, R., "Measuring Progress in Order-Verification within Software Development Projects," *Engrng. Comp.* **23**, pp. 283–294 (2007).

[Li05b] Li, S., Rider, W. J., and Shashkov, M. J., "Two-Dimensional Convergence Study for Problems with Exact Solution: Uniform and Adaptive Grids," Los Alamos National Laboratory report LA-UR-05-7985 (2005).

[Maj77] Majda, A., and Osher, S., "Propagation of error into regions of smoothness for accurate difference approximations to hyperbolic equations," *Comm. Pure Appl. Math.* **30**, pp. 671–705 (1977).

[Mar08] Margolin, L. G., and Shashkov, M. J., "Finite volume methods and the equations of finite scale: A mimetic approach," *Int. J. Num. Meth. Fluids* **56**, pp. 991–1002 (2008).

[Obe07] Oberkampf, W. L., and Trucano, T. G., "Verification and Validation Benchmarks," *Nuclear Design and Engineering* **23**, pp. 716–743 (2007); also available as Sandia National Laboratories report SAND2007-0853 (2007).

[Roa02] Roache, P., "Code Verification by the Method of Manufactured Solutions," *J. Fluids Engrng* **124**, pp. 4–10 (2002).

[Roa04] Roache, P., "Building PDE Codes to be Verifiable and Validatable," *Comput. Sci. Engrng.* **6**, pp. 30–38 (2004).

[Roy05] Roy, C. J., "Review of Code and Solution Verification Procedures for Computational Simulation", *J. Comput. Phys.* **205**, pp. 131–156 (2005).

[Sal00] Salari, K., and Knupp, P., "Code Verification by the Method of Manufactured Solutions," SAND2000-14444, June 2000.

[Sch06] Schwer, L. E. "An Overview of the PTC 60 / V&V 10 Guide for Verification and Validation in Computational Solid Mechanics," Reprint by ASME.

[Sha42] Sharpe, J. A., "The Production of Elastic Waves by Explosion Pressures. I. Theory and Empirical Field Observations," *Geophysics*, **7**, pp. 144–154 (1942).

[Ste05] Stewart, J. W., "Measures of Progress in Verification," SAND2005-4021P, (2005).

[Tim06a] Timmes, F.X., Fryxell, B., and Hrbek, G. M., *Spatial-Temporal Convergence Properties of the Tri-Lab Verification Test Suite in 1D for Code Project A*, Los Alamos National Laboratory report LA-UR-06-6444.

[Tim06b] Timmes, F. X., Fryxell, B., and Hrbek, G. M., *Two- and Three-Dimensional Properties of the Tri-Lab Verification Test Suite for Code Project A*, Los Alamos National Laboratory report LA-UR-06-6697.

[Tru03] Trucano, T. G., Pilch, M., Oberkampf, W. L., "On the Role of Code Comparisons in Verification and Validation," Sandia National Laboratories report SAND2003-2752 (2003).

[Tru06] Trucano, T. G., Swiler, L. P., Igusa, T., Oberkampf, W. L., and Pilch, M., "Calibration, validation, and sensitivity analysis: What's what," *Reliab. Engrng. Syst. Safety* **92**, pp. 1331–1357 (2006).

[Zie92] Zienkiewicz, O. C., and Zhu, J. Z., "The superconvergent patch recovery and a posteriori error estimates. Part 2: Error estimates and adaptivity," *Int. J. Num. Meth. Eng.,* **33**, pp. 1356-1382, (1992).

[Zie97] Zienkiewicz, O. C., and Taylor, R. L., The finite element patch test revisited a computer test for convergence, validation and error estimates," *Comp. Meth. Appl. Mech. Eng.,* **149**, pp. 223-254, (1996).